Locking Fast

Manuel Serrano
INRIA Mediterranée
2004 route des Lucioles - BP 93
Sophia Antipolis Cedex 06902 France
Manuel.Serrano@inria.fr

Johan Grande
Université Nice Sophia Antipolis
Les Algorithmes, Bât. Euclide B - BP 121
Sophia Antipolis Cedex 06903 France
Johan.Grande@ens-cachan.org

ABSTRACT

This article presents several independent optimizations of operations on monitors. They do not involve the low-level mutual exclusion mechanisms but rather their integration with and usage within higher-level constructs of the language. The paper reports acceleration of Hop, the Web programming language for which these optimizations have been created. The paper shows that other languages such as C and Java would also benefit from these optimizations.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Run-time environments*

Keywords

Compilation, Synchronization, Locking

1. INTRODUCTION

Preemptive scheduling and shared memory is a commonly used combination to implement parallel applications. It naturally takes advantage of physical parallelism exposed by multi-core machines but requires the programmer to identify and protect critical sections with monitors [9, 3]. Monitors have been deeply studied and the algorithms to implement them efficiently are well known [8, 6, 11]. However, the efficiency of the mechanisms for acquiring and releasing a monitor is not all. The connection with the programming language also deeply impacts the overall performances. This aspect has received less attention and huge improvements can be obtained by carefully crafting monitor usages. This is the subject of this paper. It shows three simple optimizations that significantly accelerate the applications.

Two main flavors of monitors (a.k.a. mutexes) exist in the popular mainstream programming languages. On the one hand, there is the C Posix multi-threading library [2, 4] whose API is based on the functions `pthread_mutex_lock` and `pthread_mutex_unlock`. We call this programming style *explicit locking*. On the other hand there is Java and `synchronized` blocks and synchronized methods [7]. We call this *structured locking*. We will show that, when carefully integrated in the execution environment, *structured locking* incurs no performance penalty over explicit locking.

The optimizations presented in this paper have been designed for and implemented in the Hop programming language [16]. However, they would apply equivalently well to other contexts. For instance, the observations about the IO system presented in sections 2 and 3 could be used to improve the performance of C. The implementation techniques for structured locking presented in 4 could be applied to Java and C#, as well as to the compilers of high-level languages which target C.

SAC'14 March 24-28, 2014, Gyeongju, Korea.

2. SINGLE-THREADED EXECUTION OF THREAD-SAFE CODE

In this first section we show a simple optimization concerning programs that actually run as a single thread. Let us consider the typical C program, which could have been taken from an introductory course to C.

```
#include <stdio.h>

void test( char *argv[] ) {
   FILE *f = fopen( argv[ 1 ], "w" );
   long i = atol( argv[ 2 ] );
   while( i > 0 ) fputc( i-, f );
   fclose( f );
}

int main( int argc, char *argv[] ) {
   test( argv );
}
```

More efficient alternatives exist for implementing the same behavior but this test exhibits the impact of the locking mechanism, which is the point of this section. Let us compile it with `gcc-4.7.2 -O3`. Let us link it against the gnu `glibc-2.17-1` library, and let us run it on `linux-3.6.11` executed by an Intel Xeon i7-W3570 3.2Ghz. On this machine, executing `a.out /dev/null 2000000000` lasts 17.51 seconds (best sys+cpu time of 3 consecutive runs).
The very same Hop program is:

```
(define (test argv)
  (let ((f (open-output-file (cadr argv))))
    (let loop ((i (string->integer (caddr argv))))
      (when (>fx i 0)
        (write-byte i f)
        (loop (-fx i 1))))
    (close-output-port f)))

(define (main argv)
  (test argv))
```

Functions suffixed with "fx" designated operations on small boxed integers. Hop compiles programs to C and for this simple example, the generated code is equivalent to the hand-written C version. However, the Hop version executes in only 12.27 seconds. That is 1.42 times faster than the C version. Let us explain why.

The implementations of Glibc's `fputc` and Hop's `write-byte` are almost identical as they both rely on the historical Kernighan and Richie implementation [12]. The main part of the Glibc implementation is:

```
...
if( fp->_IO_write_ptr >= fp->_IO_write_end )
   result = __overflow( fp, ch );
else
   result = (*(fp)->_IO_write_ptr++ = ch);
...
```

The variable `fp` holds the pointer to the file descriptor, which contains a buffer, a pointer to the current writing position (`_IO_write_ptr`), and a pointer to the end of the buffer (`_IO_write_end`). The fast path of the function merely stores the character and increments the writing position. When the buffer is full, it is flushed by an auxiliary function not described here. The performance difference between C and Hop does not come from any trick for managing the buffer more efficiently. It only comes from the way the two languages accommodate multi-threaded executions.

The file descriptor implements a state represented by its buffer and by the pointer `_IO_write_ptr`. If two threads call simultaneously `fputc` with the same file descriptor and if no provision is taken to enforce atomicity, the execution will be unpredictable: some characters may be lost, the execution may fail if the invariant `_IO_write_ptr < _IO_write_end` is not held, or other erratic behaviors may happen.

Although ISO C [10] assumes single-threaded executions, some C implementations ensure thread-safety for the standard IO to let multi-threaded applications to be linked against third party libraries that use `stderr` or `stdout`. To achieve this goal, in the Glibc, `fputc` is implemented as:

```
int fputc( int ch, _IO_FILE *fp ) {
   int result;

   _IO_acquire_lock( fp );
   if( fp->_IO_write_ptr >= fp->_IO_write_end )
      result = __overflow( fp, ch );
   else
      result = (*(fp)->_IO_write_ptr++ = ch);
   _IO_release_lock( fp );

   return result;
}
```

C and Hop implement `_IO_acquire_lock` differently. This, by itself, explains the performance difference. In the Glibc `_IO_acquire_lock` ends up calling the Posix function `flockfile` whatever the execution mode. When the program is executed in a singled-threaded environment, protecting the pointer increment is useless, and calling `flockfile` slows down the execution for no reason.

Hop avoids this by selecting the implementation of `_IO_acquire_lock` at runtime. Single-threaded applications use an empty function. Multi-threaded applications use an implementation that manages real monitors. The straightforward implementation consists in making `_IO_acquire_lock` a pointer to a function that is dynamically adjusted when the application starts. When executed in single-threaded environment the extra cost imposed by `_IO_acquire_lock` is thus only a computed function call, which is 1.42 times faster than the default implementation of `_IO_acquire_lock` of the Glibc. As simple as it is, this implementation would also significantly accelerate the performance of Glibc IOs of single-threaded applications.

The dynamic configuration of `_IO_acquire_lock` must be selected at the very beginning of the execution, before the user code starts to execute and before the standard files are opened, otherwise the consistency of the execution is not guaranteed. This principle is so simple that it should not be difficult to adapt to already existing systems. It is straightforward to accommodate when designing a new runtime system, as we did for Hop.

The Glibc already uses some kind of dynamic reconfiguration because a single-threaded application is faster than the same application executed in a multi-threaded context. However, instead of reconfiguring both `_IO_acquire_lock` and `flockfile` it seems that only the latter is changed. Changing both would bridge the gap between the Glibc and Hop.

Suggesting the dynamic configuration constitutes the first of the three contributions of this paper.

3. SPINLOCKS FOR IO

Now we discuss another optimization concerning multi-threaded executions. Let us modify the test to run the loop inside a thread. The new multi-threaded C version of `main` is:

```
int main( int argc, char *argv[] ) {
   pthread_t th;
   void *retval;

   pthread_create( &th, 0L, &test, argv );
   pthread_join( th, &retval );
}
```

The Hop program is modified similarly. Everything else being equal, the new C version runs in 32.63 sys+cpu seconds and the new Hop version in 24.53 sys+cpu seconds. Hop is now 1.33 times faster than C.

The Glibc implementation of `flockfile` relies on the `pthread_mutex_lock` function. This is established by observing that the following two programs run at the same speed:

```
FILE *f = ...;          | FILE *f = ...;
int c = ...;            | int c = ...;
                        | pthread_mutex_t *m = ...;
                        |
flockfile( f );         | pthread_mutex_lock( m );
putc_unlocked( c, f ); | putc_unlocked( c, f );
funlockfile( f );       | pthread_mutex_lockf( m );
```

Similarly, replacing a call to `fputc` with the sequence `pthread_mutex_lock`, `fputc_unlocked`, `pthread_mutex_unlock` leaves the performance unchanged.

Hop uses a different strategy. The implementation of `_IO_acquire_lock` depends on the nature of the file descriptor. For some file descriptors it uses full-fledged mutexes. For others, it uses spin locks, which are a faster alternative to mutexes when the contention rate is low. Hop uses an overly simple heuristics to decide which of a full-fledged mutex or a spin lock to use: it uses spin locks for file descriptors whose output operations do not usually block. That is, for regular files, the consoles, and string ports. This strategy lets Hop be 33% faster than C for writing on the standard error port, as shown by the example. This would apply equivalently well to C.

To validate the assumption that spin locks can replace mutexes, we have measured the IO contention rate of a realistic full-fledged Hop Web application, namely a distributed multimedia Web based application. This application is executed inside the Hop Web server. The GUI is executed by Web browsers. By zeroconf, the application discovers available music repositories. It generates Web pages on-the-fly for displaying various informations such as titles extracted from ID3 tags or images automatically downloaded from music databases such as musicbrainz or Last.FM. When the application plays an MP3 resource, it downloads it from the network, decodes it, and plays it on the output speaker using a low-level interface also implemented in Hop. The Hop Web server is multi-threaded. By default, it runs 20 threads which handle browsers requests in parallel. The MP3 decoder and music player are executed in two different threads synchronized by a classical producer/consumer algorithm. Various logs are generated by the Web server itself, by the MP3 decoder, and by the music player. The source code of the application spans over more than 200Kloc.

We have measured the number of spin locks acquired when using the application for *i)* browsing the music directories, *ii)* downloading a music file, and *iii)* played it during 3 minutes. During this execution, 318,727 spin locks were acquired, amongst which only 96 were busy. That is, during this execution the contention rate was only of 0.03%. This shows how well suited the spin locks are for this application. Of course a different application might exhibit a different behavior but this test is a strong indication that using spin locks instead of mutexes for protecting output operations is likely to be beneficial.

C and Hop use the same low-level locking mechanism but Hop is much faster for the presented examples that involve simple IO operations because it uses it more efficiently. This shows that the intrinsic speed of a lock implementation is not all. Its integration in the programming language also has a huge impact. Drawing the attention of compiler writers and suggesting one easy-to-implement organization for output operations constitutes the second contribution of this paper.

Optimizing monitors is a very active research field. However, most studies focus on optimizing the low level machinery for acquiring locks. This is not directly related to our study as we focus only on how to integrate these mechanisms in high-level programming languages. As it combines lightweight locks (as the *thin locks*) and full-fledged locks [1] shares some objectives and some means with our study. Thin locks and their numerous descendants rely on the idea that not all mutexes must be treated similarly. Mutexes that are never requested by two threads simultaneously could be handled more simply that those that actually protect shared data structures. In the thin lock setting, a lock starts its life as a simple compare-and-swap operation. If at some point of the execution, it happens to be requested by two threads concurrently, it is metamorphosed into a full-fledged lock, or a *fat* lock in this work terminology. Thin locks have a much more ambitious goal than our modest contribution. First, they apply to potentially all Java mutexes, the language for which they have been designed. Second, they can optimization some objects during a portion of the execution and then give up after a certain time because they involve dynamic operations. Our locking mechanism is unable to perform such sophisticated operations. It does not apply as broadly as thin locks do. However, being so simple, it has assets that might be found interesting. It demands very few modifications to an existing implementation and as such it is easy to re-use. It can also benefit to many different programming languages and systems.

4. STRUCTURED LOCKING

The C Posix locking API is based on the functions `pthread_mutex_lock` and `pthread_mutex_unlock`. No syntactic restriction limits their use but at runtime each call to `pthread_mutex_lock` must be balanced with a call to `pthread_mutex_unlock`.

The Java locking API is based on *synchronized* blocks and *synchronized* methods. Synchronized methods can be implemented with synchronized blocks so we only present the latter here. A synchronized block enforces the balance between the lock acquisition and the lock release syntactically and dynamically. No matter how the synchronized block exits, either normally or abruptly because of an exception, the monitor is always released. (Other flow-breaking mechanisms like `return` and `break` will be considered as exceptions.)

Structured locking versus explicit locking can be compared to structured programming versus programming with gotos. On the one hand explicit locking lets programmers deploy very clever and tricky implementations. On the other hand, they are difficult to use correctly and to maintain because lock and unlock can be deeply intricated. Regarding the performance, structured locking seems less efficient because of its protection against exceptions, which incurs a runtime overhead. We show in this section how to eliminate this overhead. Synchronized blocks then cumulate the comfort of a high-level programming style and the speed of a low-level mechanism.

4.1 The Usual Implementation

Java synchronized blocks have to release their monitor even when they exit abruptly because of an exception. The semantics of a synchronized Java block consists in acquiring a monitor and installing an exception handler which releases it. At the JVM byte-code level [13], this is implemented as:

```
monitorEnter( lock );
try {
  ...
} finally {
  monitorExit( lock );
}
```

Installing an exception handler is almost a free operation of the JVM. Throwing an exception is in comparison more expensive. In Hop, it is the opposite: installing an exception handler is expensive and throwing an exception relatively

cheaper. This is a consequence of the compilation strategy which uses C as intermediate language. Implementing exceptions efficiently in portable C code is difficult because `setjmp/longjmp`, the C low level mechanism for managing escapes, is not totally adequate for exceptions. The function `setjmp` saves all the arguments of the current function and potentially the signal handlers on the stack. This is slow, but only accounts for a small part of the overall performance penalty imposed by using `setjmp`. The other part comes from the saving of the temporary variables which are not directly preserved by `setjmp`. To save them, Hop introduces an extra function whose parameters are the free variables of the handler. To make the situation definitively bad, amongst these free variables, those that are mutated are boxed. Let us illustrate this complexity with an example. Let us show the compilation of the following Hop function[1]:

```
(define (foo z)
  (let ((x 1) (y 2))
     (set! y 4)
     (try-finally
        ...
        (bar (+ x y z)))))
```

The Hop function `foo` is compiled into a C function taking one integer argument. The temporary variables `x` and `y` are compiled into two C temporaries. Since `y` is mutated and used in the handler, it is boxed (the `MAKE_BOX` library function). The handler uses `x`, `y`, and `z`, which must then be saved by `setjmp`. Since `setjmp` only saves the arguments of the current function, all the variables that be must saved by a `setjmp` must be artificially implemented as function parameter. In our example, the auxiliary function `__try53` is created for this purpose. It takes the three free variables as parameters. The whole C code is:

```
obj_t __try53( int x, obj_t y, int z ) {
   jmp_buf env;
   if( setjmp( env ) == 0 )
      ...
   else
      return bar( x + BOX_REF( y ) + z );
}

function foo( int z ) {
   int x = 1;
   obj_t y = MAKE_BOX( 2 );

   BOX_SET( y, 4 );
   __try53( x, y, z );
   ...
}
```

This C code is slow, specially in comparison with the Java implementation of `try` statements. Fortunately, `synchronize` blocks can be executed without exception handlers while preserving the same semantics. With this implementation, the performance of `synchronize` block does not depend on the performance of exceptions, at all. This is explained in the next section[2].

[1]In Hop a *finally* handler is installed with a construct called `unwind-protect` but for the sake of the simplicity of the reading, this paper adopts the Java terminology.

[2]For languages that only offer exceptions to stop abruptly executions, targeting C++ instead of C and compiling exceptions into C++ exceptions is an option. Hop is not among them as it supports lexical and dynamic exists and `call/cc`.

4.2 Synchronization Lifting

The specification of synchronized blocks relies implicitly on an exception handler which enforces the release of the monitor. This handler is mandatory but it does not need to be at the syntactic location of the synchronized block. If the block completes normally the lock can be released before executing the next instruction. If the block exists abruptly because of an exception, the release of the lock can be delayed until the next `finally` or `catch` handler that will be fired by the exception is executed. Based on this observation, we can use a new compilation schema for synchronized blocks whose principle is illustrated in Figure 1. Instead of installing a dedicated exception handler on the stack for a synchronized block, the last already installed handler is marked to release the monitor when fired by an exception. We call this technique *synchronization lifting*. In Hop, the new sequence for a `synchronize` block is as follows:

```
(mutex-lock! lock)
(let ((hdl (get-current-exception-handler)))
   (handler-push-mutex! hdl lock)
   (let ((res ...))
      (handler-pop-mutex! hdl)
      (mutex-unlock! lock)
      res))
```

Two other parts of the runtime environment also need to be modified. First, the exception handlers are modified to accommodate the functions `handler-push-mutex!` and `handler-pop-mutex!`. Second, the runtime system is changed to release all the monitors that have been pushed in an exception handler, prior to executing it.

/users/serrano/diffusion/article/sac14/lifting.pdf

Figure 1: Synchronized block lifting. On the left-hand side, the regular compilation scheme. A `synchronized` block pushes on the stack its own exception handler. When an exception is thrown, the handler releases the monitor and re-throws the exception. On the right-hand side, the lifted handler. On enter, the `synchronized` block acquires the monitor, and pushes it in the current exception handler. When a exception is thrown, it directly invokes the handler which was pushed on the stack before the `synchronized` block is entered.

With this new scheme, entering a synchronized block merely pushes a mark on the stack. Leaving the block erases it. Firing an exception handler releases all pushed monitors. The fast path is almost identical to the Java `try` implementation. The slow path, *i.e.*, when the block exits abruptly, is faster than the Java implementation because it avoids unrolling a chain of `finally` blocks.

4.3 Synchronization Lifting in Hop

The Hop development kit is composed of a native compiler and an execution environment. The main element of the environment is a dedicated Web server [16] which embeds a Hop interpreter and an on-the-fly Hop-to-JavaScript client compiler [14]. The rest of the environment is composed of various libraries for multimedia applications, networking, databases, etc. The execution environment consists in 253,246 lines of Hop, the whole system being bootstrapped.

Lifted synchronized blocks execute three more operations than their explicit locking counterpart. First they retrieve the current exception handler, second, they push the lock on the stack, and third, they pop it. These last two operations can be made efficient by pre-allocating the required space in the handler. The fetching of the exception handler can be implemented as a global variable access or, when the runtime supports them, as a thread register access. The cost of these operations is hardly visible when benchmarked.

For tuning the implementation of synchronized blocks that potentially raise an exception, we have measured the number of locks that need to be pushed per exception handler. On the multimedia application described in Section 3 we have collected that 15873 synchronized blocks have been executed, in addition to the 318,727 IO spin locks which have been measured separately. Synchronization lifting stores exactly one monitor for 15720 blocks (99%). It stores two for 153 blocks (1%). It has not been observed situations where more than two monitors needed to be pushed on an exception handler. This observation yields an implementation of synchronization lifting in Hop which uses a cache of two mutexes. When more than 2 nested mutexes are locked, a list is allocated on the heap to store the extra mutexes. The C implementation is as follows:

```
struct exc_handler {
   struct exc_handler *prev;
   ...
   obj_t mutex0;
   obj_t mutex1;
   obj_t mutexN;
};

#define HOP_HANDLER_PUSH_MUTEX( exch, m ) \
   exch.mutex0 == HOP_FALSE ? exch.mutex0 = m : \
   exch.mutex1 == HOP_FALSE ? exch.mutex1 = m ) : \
      exch.mutexN, make_pair( m, exch.mutexN )

#define HOP_HANDLER_POP_MUTEX( exch ) \
   exch.mutex1 == HOP_FALSE ? \
      exch.mutex0 == HOP_FALSE ) : \
      NULLP( exch.mutexN ) ? \
         exch.mutex1 = HOP_FALSE : \
         exch.mutexN = CDR( exch.mutexN )
```

To eliminate a part of the small overhead incurred by pushing and popping monitors, the Hop native compiler deploys a simple static analysis which detects failsafe synchronized blocks, *i.e.*, synchronized blocks that do not raise exceptions. It relies on a naive per module effect analysis that traverses the AST to check if a throw or an unknown function call is reachable. Applied to the whole source code of the execution environment, this analysis detects 72 failsafe synchronized blocks out of 264, *i.e.*, 27%. At runtime, amongst the 15873 synchronized block executed by the multimedia application, 1828 (11%) are failsafe and avoid push/pop operations. This analysis could be replaced by more sophisticated ones that have been developped for Java [15, 5]; these optimizations and synchronization lifting are complementary.

4.4 Java Synchronization

Current JVM implementations such as the Oracle JVM version 6u26-1 would benefit from synchronization lifting. Synchronized blocks are implemented as plain `try/finally` blocks. When an exception is thrown, chained `finally` handlers are executed to *i)* release the monitors and to *ii)* rethrow the exception. We have measured the impact of these intermediate jumps on the overall Java performance. For that, we have implemented a Java program that runs a long loop inside a thread. At each iteration, a `try` block is executed. Depending on the version of the test, this block synchronizes several mutexes and calls an external function (**sync**), or it directly calls that function without synchronization (**unsync**). In a third version of the test, the function either directly returns (**return**) or throws an exception (**throw**). Here are the four execution times:

	sync	unsync
return	2.21s	0.63s
throw	66.89s	50.71s

The comparison of the **return**/**sync** and **return**/**unsync** executions shows that the cost of the `synchronized` blocks is about 2.21-0.63 = 1.58s. The difference between **throw**/**sync** and **throw**/**unsync** is of 16.18s. This duration minus the time needed to acquire the monitor, 16.18-1.58=14.60s, is the time needed to unroll the stack of synchronized finally handlers when the exception is thrown. Eliminating this, would be the benefit of the synchronization lifting applied to Java.

The Java JIT could lift the finally handler of a `try` statement when it only calls the `monitorExit` function. This would improve the performance of `synchronized` blocks when exceptions are thrown, while otherwise leaving the speed unchanged.

4.5 Performance Evaluation

In this section we compare the speed of three simple programs written in Hop, C, and Java. This experiment should be read as a validation of the claim that synchronization lifting reduces execution times. It should not be read as a performance comparison of languages, which is always controversial.

The first of our tests (**sync**) acquires two locks and calls a function that immediately returns. The second (**throw**) acquires two locks and calls a function that once per execution throws an exception. In C, throwing an exception is simulated by returning an error code function to function. The third test (**notify**) acquires a lock and notifies on a condition variable. The implementations are so trivial that they are not given here. The execution times of these tests are presented in Figure 2.

	Hop	Hop/Lifting	Gcc	Java
sync	38.70s	4.06s	4.08s	3.38s
throw	37.59s	4.24s	4.09s	3.33s
notify	19.56s	3.68s	3.56s	5.39s

Figure 2: Measuring the impact of Synchronization lifting on three simple tests. Reported times are obtained by summing cpu and system times. The value reported here is the minimal value observed out of three consecutive runs.

The most striking difference is between Hop and Hop/-Lifting. The acceleration obviously comes from the removal of the complex `setjmp`-based sequence. The second observation is that Hop/Lifting and Gcc deliver comparable performances. The difference between **sync** and **throw** shows the cost of pushing and popping monitors to and from the stack. The last observation is that Java is significantly faster than Hop and Gcc for the first two benchmarks and slower for the last one. This seems to show that Java trades performance of notification for the performance of the monitor acquisition. This probably reflects the flavor of Java concurrent programming style. It might also reflects that the fast path for acquiring a lock in Java is more efficient that the one of the current pthread library. Since it is beyond the scope of this paper to compare the general performance of distributed programming in Java and with pthread we did not investigate any further.

5. CONCLUSION AND FUTURE WORK

In this paper we studied the integration of low-level locking mechanisms in the programming languages execution environment. We have shown that for a given low-level locking mechanism the performance of the applications may vary significantly according to decisions taken for integrating it in the runtime system. We have studied two different aspects. First, we have shown how to accelerate C IO locking by selecting at runtime the adequate implementation and by using spin locks instead of full-fledged mutexes. Second, we have presented a new schema for improving the slow path of Java-like synchronized blocks. It consists in lifting the exception handler that is installed on the stack and which is in charge of releasing a monitor up to the closest exception handler already installed on the stack.

The synchronization lifting technique could be generalized to all the exception handlers, not only the handlers of synchronized blocks. As lifting only modifies the interception of exceptions, not the way they are thrown, it is compatible with languages such as Java or JavaScript that store a description of the stack at the moment when the exception is thrown inside the exception handlers. The technique should thus be broadly applicable. Exploring this idea is left for future work.

6. REFERENCES

[1] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for java. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 258–268, New York, NY, USA, 1998. ACM.

[2] A. Birrell. An Introduction to programming with threads. Technical Report 35, DEC Systems Research Center, Palo-Alto, CA, USA, Jan. 1989.

[3] P. Buhr, M. Fortier, and M. Coffin. Monitor Classification. *ACM Surveys*, 1(27):63–107, Mar. 1995.

[4] D. Buttlar, J. Farrell, and B. Nichols. *PThreads Programming – A POSIX Standard for Better Multiprocessing*. O'Reilly Media, 1996.

[5] D.-D. Choi, M. Gupta, M. Serrano, C. Dreedhar, and S. Midkiff. Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis. *ACM TOPLAS*, 25:2003, 2003.

[6] H. Franke, R. Russell, and M. Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium (OLS'02)*, pages 479–495, Ottawa, Canada, June 2002.

[7] J. Gosling, F. Yellin, and the Java Team. *The Java Application Programming Interface, Volume 2: Window Toolkit and Applets*. Addison Wesley, 1996.

[8] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[9] C. Hoare. Monitors: An Operating Systems Structuring Concept. *Communication of the ACM*, 10(17):549, Oct. 1974.

[10] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, Dec. 2011.

[11] K. Kawachiya, A. Koseki, and T. Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the ACM SIGPLAN OOPSLA'02 conference*, pages 130–141, New York, NY, USA, 2002.

[12] B. Kernighan and D. Richie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1978.

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine*. Addison-Wesley, 1996.

[14] F. Loitsch and M. Serrano. *Trends in Functional Programming*, volume 8, chap. 9, chapter Hop Client-Side Compilation, pages 141–158. Morazán, M. T., Seton Hall University, Intellect Bristol, UK/Chicago, USA, 2008.

[15] E. Ruf. Effective Synchronization Removal for Java. In *Proceedings of the ACM SIGPLAN PLDI'00 conference*, pages 208–218, New York, NY, USA, 2000. ACM.

[16] M. Serrano. HOP, a Fast Server for the Diffuse Web. In *proceedings of the COORDINATION'09 conference* (invited paper), Lisbon, Portugal, June 2009.

7. APPENDIX

We have measured `fputc` on three different architectures. A Linux 3.6.11 hosted by an Intel Xeon W3570, 3.2GHz, an Android 2.3.5 phone with an ARM 1 GHz Qualcomm 8255 Snapdragon running Linux 2.6.35, and a MacOS X 10.7, with a processor Intel Core i7 3720QM The test has been executed in a single threaded (**single**) environment and in a multi-threaded (**multi**) threaded environment. For the C version, an extra mode (**flock**) has been tested. Instead of using the implicit locking mechanism of `fputc`, it uses a explicit locking mechanism based of `flockfile`. The reported numbers are user+system times expressed in seconds.

	Linux x86		Android/ARM		MacOS/x86-64	
	C	**Hop**	**C**	**Hop**	**C**	**Hop**
single	17.51s	12.27s	107.10s	59s	11.75s	12.7s
multi	32.63s	24.52s	107.10s	739s	88.67s	86.42s
flock	32.63s	NA	1189s	NA	68.25s	NA

It can be observed that the three architectures deploy different strategies. Android Bionic's `fputc` is not thread-safe (**single** and **multi** have the same speed which is much higher than **flock**). Bionic does not implement spin locks so normally the Hop version of **multi** and the C **flock** should run as fast but Hop is significantly faster, without obvious rea-

son.

Apparently MacOS is fast for single-threaded environment. It is likely due to the strategy described in Section 2. Surprisingly the version **multi** is significantly slower than the **flock** version. We have found no reason for explaining this phenomenon.